Paige Rosynek

CS 3860

Dr. Magana

10.6.2022

# **Lab 3 – Data Modelling & SQL**

## **Part 1 – Create a Video Database**

##### **Review videodb-readme.txt**

After reviewing the videodb-readme.txt that contains the fields for the other data files: Video\_Categories.txt, Video\_Actors.txt, and Video\_Recordings.txt. While looking at this file, I observed that there are a lot of fields in the Video\_Recordings.txt file which could be something we have to fix later in the lab. In addition, I noticed that the Video\_Actors file has a recording\_id which indicates that there is a relationship between video actors and video recordings.

##### **Create a Video database**

Text

Description automatically generatedProof and verification of the created database is shown below.

##### **Create an SQL script to create import tables for importing the data as tab-delimited text files. Note: There are differences between importing csv- and tab-delimited files. Why would you select one format over another?**

One reason you may want to have a .csv file (which can be delimited with a ‘,’) is if you don’t know what type of system a tab-delimited file was made on, then you may have issues loading in the file because different systems have different encodings for how large a tab is. The screenshots below show the results of creating each of the tables: video\_actors, video\_categories, and video\_recordings.

Text

Description automatically generatedText

Description automatically generatedText

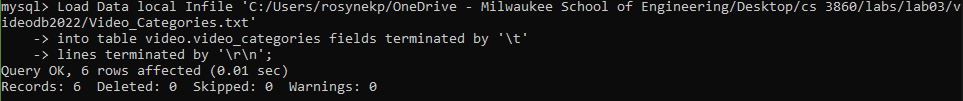
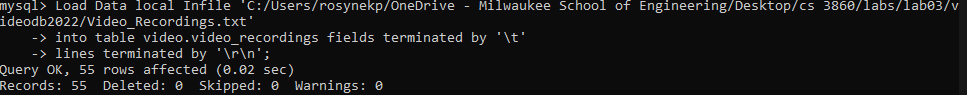
Description automatically generated

##### **Load the tables from the data files, verify the data has been successfully imported.**

The screenshots below show the ‘Load Data infile’ successful load of each of the data files. In addition, I included the ‘update’ commands that was used to clean up punctuation in the data files.

Text

Description automatically generatedGraphical user interface, text

Description automatically generated

Trimming the columns of the tables and replacing “”.

Text

Description automatically generatedText

Description automatically generatedGraphical user interface

Description automatically generatedThe screenshots below display the data loaded into each of the tables, verifying the data was loaded in correctly.

## **Part 2 – Design Your Database Schema**

##### **Relational Data Model for Database Schema**

##### Below is the relational data model for my database schema. One thing I changed from the original data is I replaced the term ‘video\_recording’ with ‘film’ for better readability. In addition, I created an actor\_film table which relates film\_ids and actor\_ids which are both foreign keys from their corresponding tables, films and actors. The actor table has an actor\_id as the primary key and then a name field. The films table has the film\_id as the primary key, the category\_id and director\_id as the foreign keys, and the following fields: title, image\_name, rating, year\_released, price, stock\_count, and duration. The films table is related to the categories table which has the category\_id as the primary key and a name field. The films and directors tables are related through the directors table. The directors table has the director\_id as the primary key and a name field.

Diagram

Description automatically generated

Relational data model for the video database.

##### **Generate SQL Script to Define/Create Tables for Schema**

The text boxes below is the SQL script created by ERDplus from the relational model above.

CREATE TABLE categories

(

category\_\_id INT NOT NULL,

name VARCHAR(100) NOT NULL,

PRIMARY KEY (category\_\_id)

);

CREATE TABLE actors

(

actor\_id INT NOT NULL,

name VARCHAR(100) NOT NULL,

PRIMARY KEY (actor\_id)

);

CREATE TABLE directors

(

director\_id INT NOT NULL,

name VARCHAR(100) NOT NULL,

PRIMARY KEY (director\_id)

);

);

CREATE TABLE films

(

film\_id INT NOT NULL,

title VARCHAR(200) NOT NULL,

image\_name VARCHAR(100) NOT NULL,

rating VARCHAR(10) NOT NULL,

year\_released INT NOT NULL,

price FLOAT NOT NULL,

stock\_count FLOAT NOT NULL,

duration FLOAT NOT NULL,

category\_\_id INT NOT NULL,

director\_id INT NOT NULL,

PRIMARY KEY (film\_id),

FOREIGN KEY (category\_\_id) REFERENCES categories(category\_\_id),

FOREIGN KEY (director\_id) REFERENCES directors(director\_id)

);

CREATE TABLE actor\_film

(

film\_id INT NOT NULL,

actor\_id INT NOT NULL,

FOREIGN KEY (film\_id) REFERENCES films(film\_id),

FOREIGN KEY (actor\_id) REFERENCES actors(actor\_id)

);

##### **Load Data from Imported Tables to Final Schema**

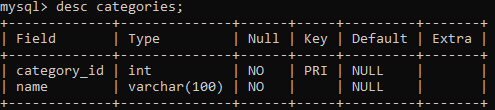
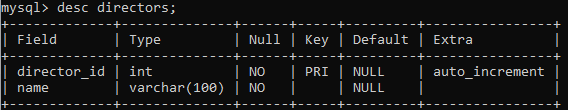
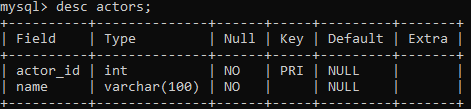
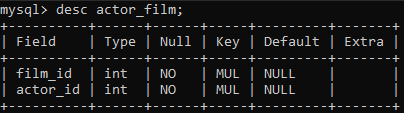
The screenshot below shows the successful creation of the tables for the new schema using the generated script above. The created tables are: actor\_film, actors, films, directors, and categories.

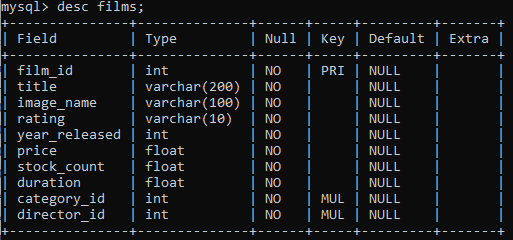
Graphical user interface, application

Description automatically generated

## **Verify Primary & Foreign Key Constraints. Why would I create the primary key index after the table has been created and the data imported versus defining the primary key in the table definition?**

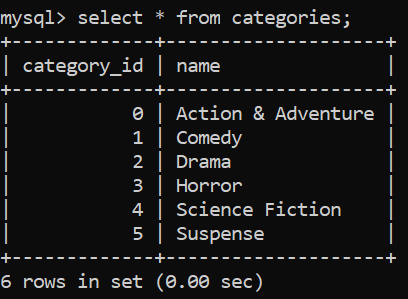
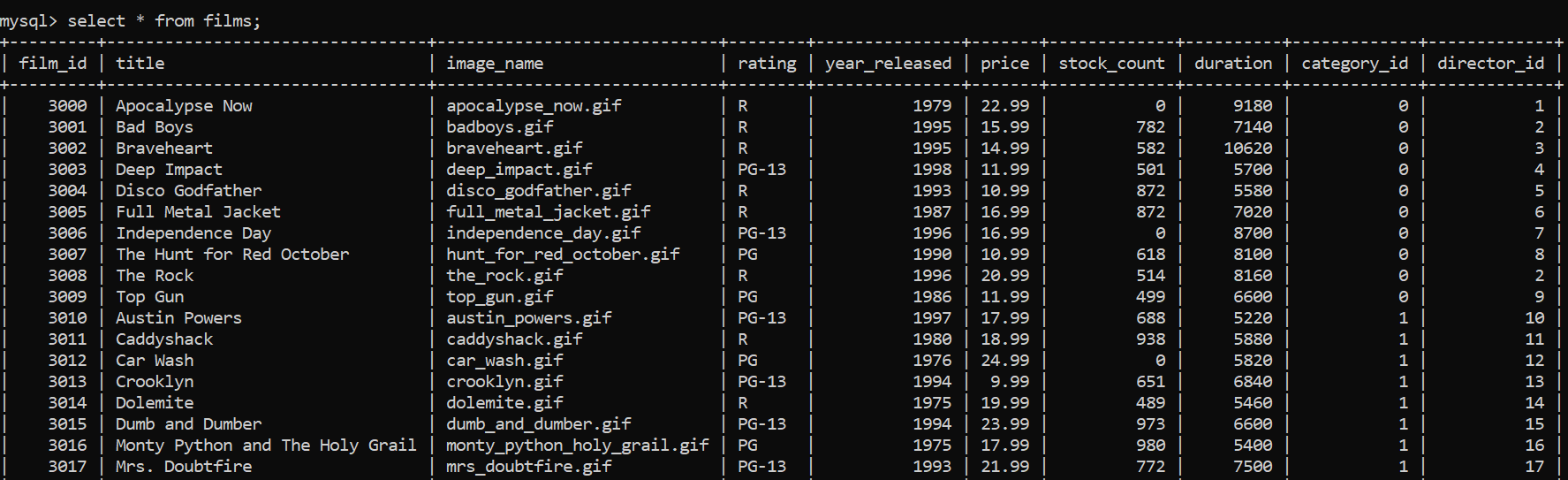
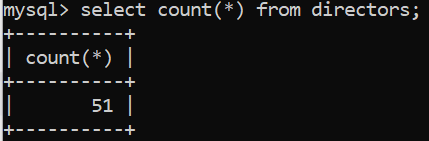
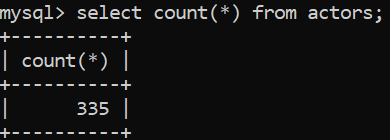
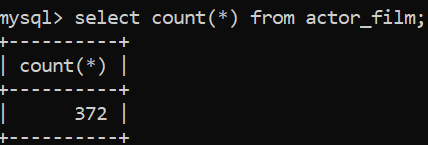
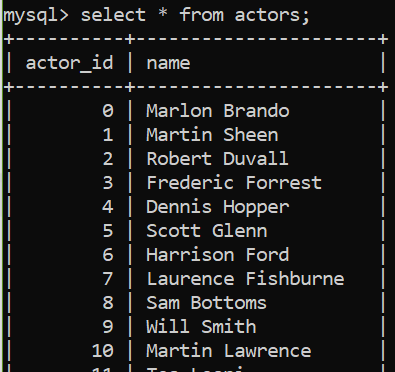
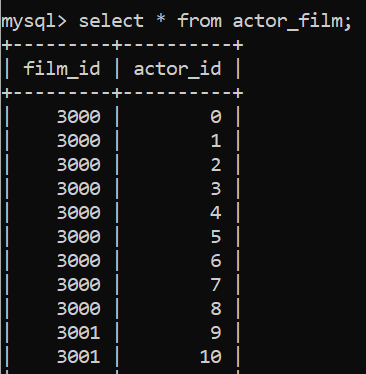
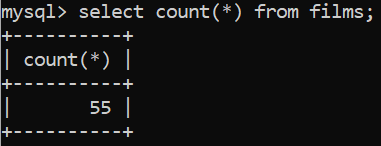
Defining the primary key after the table has been created makes inserting data into the tables easier because you won’t have to worry about getting any warnings for key constraints when loading data into the table. It can allow you to load in certain columns of data at a time as well as allow you to load tables in any order because there won’t be any key constraint errors. For my schema, I defined the key constraints of each table in the ‘CREATE TABLE’ query, however with this structure I ran into the problems described above which I solved by temporarily turning off foreign key checks which is not best practice. The screenshots below describe the primary and foreign keys of each table. The actors table has the primary key actor\_id and the film\_actors table has two foreign keys: film\_id and actor\_id. The films table has the primary key, film\_id, and two foreign keys: category\_id and director\_id. The last two tables, directors and categories, each have a single primary key, named director\_id and category\_id respectively.





## **Verify Tables Loaded**

In the screenshots below, I included the top results from a ‘select \*’ query for each of the tables, as well as the results of a count of how many rows are in each table.



### **Part 3 – SQL**

Use the original tables imported from the files for the first two questions. Include your observations and explanations along with the question, SQL, and your query results.

### **Execute:**

### **SELECT \* FROM video\_recordings, video\_categories;**

### **Note the cross-product effect of joining two tables. Record the number of rows generated. Do all permutations of Video Recordings × Video Categories make sense? Explain**

The first ten results of the above query are shown in the screenshot below; the total amount of rows returned was 330. The video\_recordings table has 55 rows and the video\_categories table has 6 rows. The above query takes each row of the video\_recordings table and joins it with each row of the video\_categories table, so each of the 55 rows will be printed 6 times, thus resulting in 55 \* 6 = 330 rows total. From inspection, the permutations of video\_recordings x video\_categories does not make sense. For each of the rows in video\_recordings, each row of video\_categories is tacked on the end. This does not make sense because a single movie cannot belong to every category as shown in the first 6 rows of the screenshot below.

Text

Description automatically generated with low confidence

### **Execute:**

**SELECT \***

**FROM video\_recordings vr, video\_categories vc**

**WHERE vr.category = vc.name;**

### **Note the cross-product effect of joining two tables when restricted on the appropriate keys. Record the number of rows generated. Explain the purpose of the join.**

Graphical user interface, text

Description automatically generated with medium confidenceThe first ten results of the above query are shown in the screenshot below; the total amount of rows returned was 55 which is equal to number of rows in the video\_recordings table. The purpose of the above query is to join the two tables where the category column of video\_recordings is matches the name column of video\_categories.

### **For the remaining questions, use your relational schema.**

### **List the number of videos for each video category.**

Query:

SELECT c.name, f.category\_id, COUNT(f.film\_id) AS num\_videos

FROM films f, categories c

WHERE c.category\_id = f.category\_id

GROUP BY f.category\_id;

A picture containing text

Description automatically generated Results:

The above query uses the films table by grouping by each category\_id and then counting the number of film\_ids in each group. The category table is used to get the category name that corresponds with each category\_id. The query returned a total of six rows.

### **List the number of videos for each video category where the inventory is non-zero**

Query:

SELECT c.name, f.category\_id, COUNT(f.film\_id) AS num\_videos

FROM films f, categories c

WHERE c.category\_id = f.category\_id AND f.stock\_count != 0

GROUP BY f.category\_id;

A picture containing text

Description automatically generatedResults:

The above query is the doing the same thing as the query in #3, however, the above query adds another constraint to the WHERE clause to filter the data such that the stock\_count for the film was nonzero. These results make sense because the number of videos for each categories is less than the total film counts in #3. The query returned a total of six rows.

### **For each actor, list the video categories that actor has appeared in.**

Query:

SELECT a.name, GROUP\_CONCAT(c.name)

FROM actors a, films f, categories c, actor\_film af

WHERE af.actor\_id = a.actor\_id

AND af.film\_id = f.film\_id

AND f.category\_id = c.category\_id

GROUP BY a.name;

Table

Description automatically generatedResults:

The screenshot above shows the first ten results of the query; the query returned a total of 335 rows which makes sense because that is the number of distinct actors in the database. The above query joins the tables: actors, films, actor\_film, and categories, then it filters the data to get the records in with matching actor\_ids, film\_ids, and category\_ids to get the categories of each of the films that each actor has been in. Then it concatenates the different categories each actor has been in.

### **Which actors have appeared in movies in different video categories?**

Query:

SELECT a.name, c.name

FROM actors a, films f, categories c, actor\_film af

WHERE af.actor\_id = a.actor\_id

AND af.film\_id = f.film\_id

AND f.category\_id = c.category\_id

GROUP BY a.name

HAVING COUNT(\*) > 1;

Results:

I spent a lot of time trying to find a query that would answer the question, but the closest thing I could come up with (it does not return any results. I attempted to filter the data to find all the categories that each actgor has been in and then group those results by each actor and then only choose the actors that had more than one category.

### **Which actors have not appeared in a comedy?**

Query:

SELECT a.name, c.name

FROM actors a, films f, categories c, actor\_film af

WHERE a.actor\_id = af.actor\_id

AND af.film\_id = f.film\_id

AND f.category\_id = c.category\_id

HAVING c.name != "Comedy";

Table

Description automatically generated Results:

The screenshot above shows the first ten results of the above query; the total number of row returned by the query was 269. The query first gets the corresponding records from the actors, actor\_film, films, and categories tables to get the categories that each actor has been in. Then it filters the table further to only get the actors that have corresponding categories that are not ‘Comedy’.

### **Which actors have appeared in both a comedy and an action adventure movie?**

Query:

SELECT a.name

FROM actors a, films f, categories c, actor\_film af

WHERE (a.actor\_id = af.actor\_id

AND af.film\_id = f.film\_id

AND f.category\_id = c.category\_id)

AND (c.name = "Comedy" OR c.name = "Action & Adventure")

GROUP BY a.actor\_id

HAVING COUNT(\*) > 1;

Results:

Unfortunately, I was unable to write a query to answer this question. The query above was as close as I could get but the query itself does not return anything. In theory, this query should filter all of the tables to match actors to the categories they have been in, and also filter for only the actors that have been in a “Comedy” or an “Action & Adventure”. Then it would group the actors by their ids and then only take the actors that had a count of greater than 1 with would mean that the actor was in both a comedy and an action & adventure film.

### **Final ERD:**

Diagram

Description automatically generated

### **Note to the professor**

I apologize for how late I turned this lab in. Between studying for the exam and studying for other classes I couldn’t get it done during the week. Also, some of my queries don’t work and I spent as much time as I could trying to get them to work without luck; I planned on having more time to work on it on Saturday but there was a scheduling error with my job and I ended up having to work for much longer than anticipated. I know this doesn’t excuse my work, but I would appreciate it if I could meet about where I went wrong in the lab so I can understand my mistakes.